

BOOK 6: OBJECTS

Chapter 1: Creating Objects

WHAT IS AN OBJECT

An object is a collection of related data and functionality. It groups properties (data) and methods (functions) together.

```
let person = {  
  name: "Alice",  
  age: 25,  
  city: "New York"  
}
```

This object represents a person with three properties: name, age, and city.

Arrays store ordered lists of similar things:

```
let names = ["Alice", "Bob", "Charlie"]
```

Objects store named properties of different things:

```
let person = {  
  name: "Alice",  
  age: 25,  
  isStudent: true  
}
```

OBJECT LITERAL SYNTAX

The most common way to create objects:

```
let car = {  
  brand: "Toyota",
```

```
model: "Camry",  
year: 2022,  
color: "blue"  
}
```

Properties are key-value pairs separated by commas.
Each property has a name (key) and a value.
The object is enclosed in curly braces.

Empty object:

```
let empty = {}
```

Add properties later.

ACCESSING PROPERTIES

Dot notation (most common):

```
let person = {  
  name: "Alice",  
  age: 25  
}
```

```
console.log(person.name)  
console.log(person.age)
```

Output:

```
Alice  
25
```

Bracket notation:

```
console.log(person["name"])  
console.log(person["age"])
```

Same output.

Bracket notation is required when:

Property name has spaces or special characters:

```
let obj = {  
  "first name": "Alice",  
  "my-property": 123  
}
```

```
console.log(obj["first name"])  
console.log(obj["my-property"])
```

Property name is in a variable:

```
let key = "name"  
console.log(person[key])
```

Output: Alice

MODIFYING PROPERTIES

Change existing property:

```
let person = {  
  name: "Alice",  
  age: 25  
}
```

```
person.age = 26
```

```
console.log(person.age)
```

Output: 26

Add new property:

```
person.city = "New York"
```

```
console.log(person.city)
```

Output: New York

Delete property:

```
delete person.city
```

```
console.log(person.city)
```

Output: undefined

PROPERTY SHORTHAND

When variable name matches property name:

```
let name = "Alice"
```

```
let age = 25
```

```
let person = {  
  name: name,  
  age: age  
}
```

Shorthand:

```
let person = {  
  name,  
  age  
}
```

Same result. Less repetition.

NESTED OBJECTS

Objects can contain other objects:

```
let person = {  
  name: "Alice",  
  age: 25,  
  address: {  
    street: "123 Main St",  
    city: "New York",  
    zip: "10001"  
  }  
}
```

```
console.log(person.address.city)
```

Output: New York

Access nested properties with chained dots.

Modify nested property:

```
person.address.zip = "10002"
```

OBJECTS WITH ARRAYS

Objects can contain arrays:

```
let person = {  
  name: "Alice",  
  hobbies: ["reading", "gaming", "cooking"]  
}
```

```
console.log(person.hobbies[0])
```

Output: reading

Arrays can contain objects:

```
let people = [  
  { name: "Alice", age: 25 },  
  { name: "Bob", age: 30 }  
]
```

```
console.log(people[0].name)
```

Output: Alice

CHECKING IF PROPERTY EXISTS

Using in operator:

```
let person = {  
  name: "Alice",  
  age: 25  
}
```

```
console.log("name" in person)  
console.log("email" in person)
```

Output:

```
true  
false
```

Using hasOwnProperty:

```
console.log(person.hasOwnProperty("name"))  
console.log(person.hasOwnProperty("email"))
```

Output:

```
true  
false
```

Checking for undefined:

```
if (person.email !== undefined) {  
  console.log(person.email)  
} else {  
  console.log("No email")  
}
```

COMPUTED PROPERTY NAMES

Property names from expressions:

```
let key = "color"
```

```
let obj = {  
  [key]: "red"  
}
```

```
console.log(obj.color)
```

Output: red

Dynamic property names:

```
let prefix = "user"
```

```
let obj = {  
  [prefix + "_name"]: "Alice",  
  [prefix + "_age"]: 25  
}
```

```
console.log(obj.user_name)
```

Output: Alice

OBJECT CONSTRUCTOR

Using `new Object()`:

```
let person = new Object()
```

```
person.name = "Alice"
```

```
person.age = 25
```

Rarely used. Object literals are preferred.

Using `Object.create()`:

```
let personProto = {  
  greet() {  
    console.log(`Hello, I am ${this.name}`)  
  }  
}
```

```
let person = Object.create(personProto)
```

```
person.name = "Alice"
```

```
person.greet()
```

Output: Hello, I am Alice

More on prototypes in a later chapter.

PRACTICAL EXAMPLES

User profile:

```
let user = {  
  username: "alice123",  
  email: "alice@example.com",  
  profile: {
```

```
firstName: "Alice",
lastName: "Smith",
avatar: "avatar.jpg"
},
settings: {
  darkMode: true,
  notifications: true
}
}
```

Product:

```
let product = {
  id: "SKU-001",
  name: "Laptop",
  price: 999.99,
  category: "Electronics",
  inStock: true,
  specs: {
    brand: "Dell",
    processor: "Intel i7",
    ram: "16GB"
  }
}
```

Configuration:

```
let config = {
  apiUrl: "https://api.example.com",
  timeout: 5000,
  retries: 3,
  debug: false
}
```

PRACTICE

Create an object representing a book with title, author, and year.

Access properties using both dot and bracket notation.

Add a new property to an existing object.

Create a nested object with address information.

Check if a property exists on an object.

WHAT YOU LEARNED

You learned that objects group related data as properties.

You learned to create objects with curly braces.

You learned dot notation and bracket notation.

You learned to add, modify, and delete properties.

You learned about nested objects.

You learned to check if properties exist.

The next chapter teaches you about object methods.

BOOK 6: OBJECTS

Chapter 2: Object Methods

METHODS ARE FUNCTIONS IN OBJECTS

A method is a function that belongs to an object:

```
let person = {  
  name: "Alice",  
  age: 25,  
  greet: function() {  
    console.log("Hello!")  
  }  
}
```

```
person.greet()
```

Output: Hello!

The greet property holds a function. Calling person.greet() runs that function.

METHOD SHORTHAND

Modern JavaScript has a shorter syntax:

```
let person = {  
  name: "Alice",  
  greet() {  
    console.log("Hello!")  
  }  
}
```

```
person.greet()
```

Remove the colon and function keyword. Same behavior.

THE THIS KEYWORD

Inside a method, this refers to the object:

```
let person = {  
  name: "Alice",  
  age: 25,  
  introduce() {  
    console.log(`Hi, I am ${this.name}`)  
    console.log(`I am ${this.age} years old`)  
  }  
}
```

```
}
```

```
person.introduce()
```

Output:

Hi, I am Alice

I am 25 years old

this.name accesses the name property of the same object.

this.age accesses the age property.

Without this:

```
let person = {  
  name: "Alice",  
  introduce() {  
    console.log(`Hi, I am ${name}`)  
  }  
}
```

```
person.introduce()
```

Error or wrong value. name looks for a variable, not the property.

METHODS THAT RETURN VALUES

```
let calculator = {  
  add(a, b) {  
    return a + b  
  },  
  subtract(a, b) {  
    return a - b  
  },  
  multiply(a, b) {  
    return a * b  
  }  
}
```

```
console.log(calculator.add(5, 3))
console.log(calculator.multiply(4, 7))
```

Output:

```
8
28
```

METHODS THAT MODIFY THE OBJECT

```
let counter = {
  value: 0,
  increment() {
    this.value++
  },
  decrement() {
    this.value--
  },
  reset() {
    this.value = 0
  }
}
```

```
console.log(counter.value)
counter.increment()
counter.increment()
console.log(counter.value)
counter.reset()
console.log(counter.value)
```

Output:

```
0
2
0
```

METHODS CALLING OTHER METHODS

Use this to call another method:

```
let person = {  
  firstName: "Alice",  
  lastName: "Smith",  
  getFullName() {  
    return `${this.firstName} ${this.lastName}`  
  },  
  introduce() {  
    console.log(`Hello, I am ${this.getFullName()}`)  
  }  
}
```

```
person.introduce()
```

Output: Hello, I am Alice Smith

`this.getFullName()` calls the `getFullName` method from `introduce`.

THIS IN DIFFERENT CONTEXTS

this depends on how the method is called:

```
let person = {  
  name: "Alice",  
  greet() {  
    console.log(`Hello, I am ${this.name}`)  
  }  
}
```

```
person.greet()
```

```
let greetFunction = person.greet
```

```
greetFunction()
```

Output:

Hello, I am Alice

Hello, I am undefined

When called as `person.greet()`, this is the person object.

When called as `greetFunction()`, this is not the person object.

This is a common source of bugs. Be careful when passing methods around.

ARROW FUNCTIONS AND THIS

Arrow functions do not have their own this:

```
let person = {  
  name: "Alice",  
  greetRegular: function() {  
    console.log(`Regular: ${this.name}`)  
  },  
  greetArrow: () => {  
    console.log(`Arrow: ${this.name}`)  
  }  
}
```

```
person.greetRegular()
```

```
person.greetArrow()
```

Output:

Regular: Alice

Arrow: undefined

The arrow function's `this` is the outer scope, not the object.

Use regular functions or method shorthand for object methods.

PRACTICAL EXAMPLES

Bank account:

```
let account = {  
  balance: 1000,  
  deposit(amount) {  
    if (amount > 0) {  
      this.balance += amount  
      console.log(`Deposited ${amount}`)  
    }  
  },  
  withdraw(amount) {  
    if (amount > 0 && amount <= this.balance) {  
      this.balance -= amount  
      console.log(`Withdrew ${amount}`)  
      return true  
    }  
    console.log("Insufficient funds")  
    return false  
  },  
  getBalance() {  
    return this.balance  
  }  
}
```

```
account.deposit(500)  
account.withdraw(200)  
console.log(`Balance: ${account.getBalance()}`)
```

Player character:

```
let player = {  
  name: "Hero",  
  health: 100,  
  maxHealth: 100,  
  level: 1,
```

```

experience: 0,
takeDamage(amount) {
  this.health = Math.max(0, this.health - amount)
  console.log(` ${this.name} took ${amount} damage. Health:
${this.health}`)
  if (this.health === 0) {
    console.log(` ${this.name} has been defeated!`)
  }
},
heal(amount) {
  this.health = Math.min(this.maxHealth, this.health + amount)
  console.log(` ${this.name} healed. Health: ${this.health}`)
},
gainExperience(amount) {
  this.experience += amount
  if (this.experience >= 100) {
    this.levelUp()
  }
},
levelUp() {
  this.level++
  this.experience -= 100
  this.maxHealth += 10
  this.health = this.maxHealth
  console.log(`Level up! Now level ${this.level}`)
}
}

```

```

player.takeDamage(30)
player.heal(20)
player.gainExperience(100)

```

Shopping cart:

```

let cart = {
  items: [],
  addItem(name, price, quantity = 1) {
    let existing = this.items.find(item => item.name === name)
    if (existing) {
      existing.quantity += quantity
    }
  }
}

```

```

    } else {
      this.items.push({ name, price, quantity })
    }
  },
  removeItem(name) {
    this.items = this.items.filter(item => item.name !== name)
  },
  getTotal() {
    return this.items.reduce((total, item) => {
      return total + (item.price * item.quantity)
    }, 0)
  },
  getItemCount() {
    return this.items.reduce((count, item) => count + item.quantity,
0)
  },
  clear() {
    this.items = []
  }
}

```

```

cart.addItem("Apple", 1.50, 3)
cart.addItem("Banana", 0.75, 5)
console.log(`Items: ${cart.getItemCount()}`)
console.log(`Total: $$${cart.getTotal().toFixed(2)}`)

```

GETTERS AND SETTERS

Special methods that look like properties:

```

let circle = {
  radius: 5,
  get diameter() {
    return this.radius * 2
  },
  set diameter(value) {
    this.radius = value / 2
  }
}

```

```
},  
get area() {  
  return Math.PI * this.radius * this.radius  
}  
}
```

```
console.log(circle.radius)  
console.log(circle.diameter)  
console.log(circle.area)
```

```
circle.diameter = 20
```

```
console.log(circle.radius)
```

Output:

```
5  
10  
78.54...  
10
```

Getters run when you read the property.

Setters run when you assign to the property.

PRACTICE

Create an object with a method that uses this.

Create a counter object with increment and decrement methods.

Create an object where one method calls another method.

Add a getter that calculates a derived value.

WHAT YOU LEARNED

You learned that methods are functions inside objects.
You learned the method shorthand syntax.
You learned that `this` refers to the object inside methods.
You learned to use `this` to access properties and other methods.
You learned that arrow functions do not have their own `this`.
You learned about getters and setters.

The next chapter teaches you how to loop through object properties.

BOOK 6: OBJECTS

Chapter 3: Looping Through Objects

OBJECTS ARE NOT ITERABLE

You cannot use `for...of` directly on objects:

```
let person = {  
  name: "Alice",  
  age: 25,  
  city: "New York"  
}
```

```
for (let item of person) {  
  console.log(item)  
}
```

Error. Objects are not iterable like arrays.

You need to use special methods or `for...in`.

FOR...IN LOOP

The for...in loop iterates over property names:

```
let person = {  
  name: "Alice",  
  age: 25,  
  city: "New York"  
}
```

```
for (let key in person) {  
  console.log(key)  
}
```

Output:

```
name  
age  
city
```

To get values, use bracket notation:

```
for (let key in person) {  
  console.log(`${key}: ${person[key]}`)  
}
```

Output:

```
name: Alice  
age: 25  
city: New York
```

OBJECT.KEYS()

Returns an array of property names:

```
let person = {  
  name: "Alice",  
  age: 25,  
  city: "New York"  
}
```

```
let keys = Object.keys(person)
```

```
console.log(keys)
```

Output: ["name", "age", "city"]

Now you can use array methods:

```
Object.keys(person).forEach(key => {  
  console.log(` ${key}: ${person[key]}`)  
})
```

OBJECT.VALUES()

Returns an array of property values:

```
let person = {  
  name: "Alice",  
  age: 25,  
  city: "New York"  
}
```

```
let values = Object.values(person)
```

```
console.log(values)
```

Output: ["Alice", 25, "New York"]

Sum all numeric values:

```
let scores = {  
  math: 85,  
  english: 92,  
  science: 78  
}
```

```
let total = Object.values(scores).reduce((sum, score) => sum + score, 0)
```

```
console.log(`Total: ${total}`)
```

Output: Total: 255

OBJECT.ENTRIES()

Returns an array of [key, value] pairs:

```
let person = {  
  name: "Alice",  
  age: 25,  
  city: "New York"  
}
```

```
let entries = Object.entries(person)
```

```
console.log(entries)
```

Output:

```
[  
  ["name", "Alice"],  
  ["age", 25],  
  ["city", "New York"]  
]
```

Use with destructuring:

```
for (let [key, value] of Object.entries(person)) {  
  console.log(`${key}: ${value}`)  
}
```

Output:

```
name: Alice  
age: 25
```

city: New York

Use with array methods:

```
Object.entries(person)
  .filter(([key, value]) => typeof value === "string")
  .forEach(([key, value]) => {
    console.log(`${key} is a string: ${value}`)
  })
```

OBJECT.FROMENTRIES()

Convert entries back to an object:

```
let entries = [
  ["name", "Alice"],
  ["age", 25]
]

let person = Object.fromEntries(entries)

console.log(person)
```

Output: { name: "Alice", age: 25 }

Useful for transforming objects:

```
let prices = {
  apple: 1.50,
  banana: 0.75,
  orange: 2.00
}

let doubled = Object.fromEntries(
  Object.entries(prices).map(([key, value]) => [key, value * 2])
)
```

```
console.log(doubled)
```

Output: { apple: 3, banana: 1.5, orange: 4 }

COMPARING METHODS

for...in: Loops through keys

Object.keys(): Array of keys

Object.values(): Array of values

Object.entries(): Array of [key, value] pairs

Use Object methods when you need array functionality.

Use for...in for simple iteration.

PRACTICAL EXAMPLES

Count properties:

```
let obj = { a: 1, b: 2, c: 3, d: 4, e: 5 }
```

```
let count = Object.keys(obj).length
```

```
console.log(`Properties: ${count}`)
```

Output: Properties: 5

Find property by value:

```
let colors = {  
  red: "#FF0000",  
  green: "#00FF00",  
  blue: "#0000FF"  
}
```

```
function findKeyByValue(obj, value) {  
  return Object.keys(obj).find(key => obj[key] === value)  
}
```

```
console.log(findKeyByValue(colors, "#00FF00"))
```

Output: green

Filter object properties:

```
let data = {  
  name: "Alice",  
  age: 25,  
  password: "secret",  
  email: "alice@example.com",  
  token: "abc123"  
}
```

```
let sensitiveKeys = ["password", "token"]
```

```
let safe = Object.fromEntries(  
  Object.entries(data).filter(([key]) => !  
    sensitiveKeys.includes(key))  
)
```

```
console.log(safe)
```

Output: { name: "Alice", age: 25, email: "alice@example.com" }

Transform values:

```
let temps = {  
  monday: 72,  
  tuesday: 75,  
  wednesday: 68  
}
```

```
let tempsCelsius = Object.fromEntries(  
  Object.entries(temps).map(([day, f]) => [day, Math.round((f -  
    32) * 5/9)])
```

)

```
console.log(tempsCelsius)
```

Output: { monday: 22, tuesday: 24, wednesday: 20 }

Group by property:

```
let students = [  
  { name: "Alice", grade: "A" },  
  { name: "Bob", grade: "B" },  
  { name: "Charlie", grade: "A" },  
  { name: "Diana", grade: "B" }  
]
```

```
let byGrade = students.reduce((acc, student) => {  
  let grade = student.grade  
  if (!acc[grade]) {  
    acc[grade] = []  
  }  
  acc[grade].push(student.name)  
  return acc  
, {})
```

```
console.log(byGrade)
```

Output: { A: ["Alice", "Charlie"], B: ["Bob", "Diana"] }

FOR...IN AND INHERITED PROPERTIES

for...in loops through inherited properties too:

```
let parent = { inherited: true }  
let child = Object.create(parent)  
child.own = true
```

```
for (let key in child) {
```

```
console.log(key)
}
```

Output:
own
inherited

Use `hasOwnProperty` to filter:

```
for (let key in child) {
  if (child.hasOwnProperty(key)) {
    console.log(key)
  }
}
```

Output: own

`Object.keys` only returns own properties:

```
console.log(Object.keys(child))
```

Output: ["own"]

PRACTICE

Loop through an object and print all key-value pairs.

Get an array of all property names from an object.

Calculate the sum of all numeric values in an object.

Transform an object by doubling all values.

Filter an object to keep only certain properties.

WHAT YOU LEARNED

You learned that `for...in` loops through property names.

You learned `Object.keys` returns an array of keys.

You learned `Object.values` returns an array of values.

You learned `Object.entries` returns `[key, value]` pairs.

You learned `Object.fromEntries` creates an object from entries.

You learned to transform and filter objects.

The next chapter teaches you about copying and comparing objects.

BOOK 6: OBJECTS

Chapter 4: Copying and Comparing Objects

OBJECTS ARE REFERENCES

When you assign an object to a variable, you assign a reference:

```
let person1 = { name: "Alice", age: 25 }
```

```
let person2 = person1
```

```
person2.age = 30
```

```
console.log(person1.age)
```

```
console.log(person2.age)
```

Output:

```
30
```

```
30
```

Both variables point to the same object. Changing one changes both.

This is different from primitive values:

```
let a = 5  
let b = a  
b = 10
```

```
console.log(a)  
console.log(b)
```

Output:

```
5  
10
```

Numbers, strings, and booleans are copied by value. Objects are copied by reference.

COMPARING OBJECTS

Two objects with the same content are not equal:

```
let obj1 = { name: "Alice" }  
let obj2 = { name: "Alice" }  
  
console.log(obj1 === obj2)  
console.log(obj1 == obj2)
```

Output:

```
false  
false
```

They are different objects in memory, even if they look identical.

Same object is equal:

```
let obj1 = { name: "Alice" }  
let obj2 = obj1  
  
console.log(obj1 === obj2)
```

Output: true

Both point to the same object.

SHALLOW COPY

A shallow copy creates a new object with the same top-level properties:

Using spread operator:

```
let original = { name: "Alice", age: 25 }
```

```
let copy = { ...original }
```

```
copy.age = 30
```

```
console.log(original.age)
```

```
console.log(copy.age)
```

Output:

25

30

Now they are separate objects.

Using Object.assign:

```
let copy = Object.assign({}, original)
```

Same result.

SHALLOW COPY LIMITATION

Nested objects are still shared:

```
let original = {  
  name: "Alice",  
  address: {  
    city: "New York"  
  }  
}
```

```
let copy = { ...original }
```

```
copy.address.city = "Boston"
```

```
console.log(original.address.city)  
console.log(copy.address.city)
```

Output:

Boston

Boston

The nested address object is shared. Shallow copy only copies one level deep.

DEEP COPY

A deep copy copies all levels of nesting.

Using JSON (simple but has limitations):

```
let original = {  
  name: "Alice",  
  address: {  
    city: "New York"  
  }  
}
```

```
let copy = JSON.parse(JSON.stringify(original))
```

```
copy.address.city = "Boston"
```

```
console.log(original.address.city)  
console.log(copy.address.city)
```

Output:

New York

Boston

Now they are completely separate.

JSON limitations:

- Cannot copy functions
- Cannot copy undefined
- Cannot copy Date objects (becomes strings)
- Cannot handle circular references

Using structuredClone (modern):

```
let copy = structuredClone(original)
```

This is the recommended modern way. It handles most cases.

COMPARING OBJECT CONTENTS

To compare if two objects have the same content:

Simple comparison function:

```
function shallowEqual(obj1, obj2) {  
  let keys1 = Object.keys(obj1)  
  let keys2 = Object.keys(obj2)  
  
  if (keys1.length !== keys2.length) {  
    return false  
  }  
}
```

```
for (let key of keys1) {  
  if (obj1[key] !== obj2[key]) {  
    return false  
  }  
}  
  
return true  
}
```

```
let a = { name: "Alice", age: 25 }  
let b = { name: "Alice", age: 25 }  
let c = { name: "Bob", age: 30 }
```

```
console.log(shallowEqual(a, b))  
console.log(shallowEqual(a, c))
```

Output:

true

false

Using JSON (for deep comparison):

```
function deepEqual(obj1, obj2) {  
  return JSON.stringify(obj1) === JSON.stringify(obj2)  
}
```

Simple but has the same limitations as JSON copy.

MERGING OBJECTS

Combine multiple objects:

Using spread:

```
let defaults = { color: "blue", size: "medium" }  
let userPrefs = { size: "large" }
```

```
let settings = { ...defaults, ...userPrefs }
```

```
console.log(settings)
```

Output: { color: "blue", size: "large" }

Later spreads override earlier ones.

Using Object.assign:

```
let settings = Object.assign({}, defaults, userPrefs)
```

Same result.

Object.assign modifies the first argument:

```
Object.assign(defaults, userPrefs)
```

```
console.log(defaults)
```

Output: { color: "blue", size: "large" }

defaults is modified. Use an empty object as first argument to avoid this.

PRACTICAL EXAMPLES

Clone and modify:

```
let original = {  
  name: "Alice",  
  scores: [85, 90, 95]  
}
```

```
let modified = {  
  ...original,
```

```
scores: [...original.scores, 100]
}
```

```
console.log(original.scores.length)
console.log(modified.scores.length)
```

Output:

3

4

Update nested property immutably:

```
let user = {
  name: "Alice",
  profile: {
    avatar: "default.jpg",
    bio: "Hello"
  }
}
```

```
let updated = {
  ...user,
  profile: {
    ...user.profile,
    avatar: "new.jpg"
  }
}
```

```
console.log(user.profile.avatar)
console.log(updated.profile.avatar)
```

Output:

default.jpg

new.jpg

Merge configurations:

```
function getConfig(userConfig = {}) {
  let defaults = {
    theme: "light",
```

```
language: "en",
notifications: true,
autoSave: true
}

return { ...defaults, ...userConfig }
}
```

```
console.log(getConfig({ theme: "dark" })))
```

Output: { theme: "dark", language: "en", notifications: true, autoSave: true }

Check if object changed:

```
function hasChanged(original, current) {
  return JSON.stringify(original) !== JSON.stringify(current)
}
```

```
let saved = { name: "Alice", age: 25 }
let form = { name: "Alice", age: 25 }
```

```
console.log(hasChanged(saved, form))
```

```
form.age = 26
```

```
console.log(hasChanged(saved, form))
```

Output:
false
true

FREEZING OBJECTS

Prevent modifications:

```
let config = Object.freeze({
```

```
apiUrl: "https://api.example.com",  
timeout: 5000  
})
```

```
config.timeout = 10000
```

```
console.log(config.timeout)
```

Output: 5000

The assignment silently fails. In strict mode, it throws an error.

Freeze is shallow:

```
let obj = Object.freeze({  
  nested: { value: 1 }  
})
```

```
obj.nested.value = 2
```

```
console.log(obj.nested.value)
```

Output: 2

Nested objects are not frozen.

PRACTICE

Create a shallow copy of an object using spread.

Create a deep copy using JSON or structuredClone.

Write a function that compares two objects for equality.

Merge two configuration objects.

Freeze an object and try to modify it.

WHAT YOU LEARNED

You learned that objects are assigned by reference.
You learned that `===` compares references, not content.
You learned shallow copy with `spread` and `Object.assign`.
You learned deep copy with `JSON` or `structuredClone`.
You learned to compare object contents.
You learned to merge objects.
You learned to freeze objects.

The next chapter teaches you about destructuring objects.

BOOK 6: OBJECTS

Chapter 5: Destructuring Objects

EXTRACTING PROPERTIES

Destructuring extracts values from objects into variables:

```
let person = {  
  name: "Alice",  
  age: 25,  
  city: "New York"  
}
```

```
let { name, age, city } = person
```

```
console.log(name)  
console.log(age)  
console.log(city)
```

Output:
Alice

25

New York

Instead of:

```
let name = person.name
```

```
let age = person.age
```

```
let city = person.city
```

One line does it all.

DIFFERENT VARIABLE NAMES

Use colons to rename:

```
let person = {
```

```
  name: "Alice",
```

```
  age: 25
```

```
}
```

```
let { name: personName, age: personAge } = person
```

```
console.log(personName)
```

```
console.log(personAge)
```

Output:

Alice

25

The property name comes first, then the variable name.

DEFAULT VALUES

Provide defaults for missing properties:

```
let person = {  
  name: "Alice"  
}
```

```
let { name, age = 30 } = person
```

```
console.log(name)  
console.log(age)
```

Output:

Alice

30

age was missing, so it got the default value.

Combine with renaming:

```
let { name: personName = "Unknown", age: personAge = 0 } =  
person
```

NESTED DESTRUCTURING

Extract from nested objects:

```
let user = {  
  name: "Alice",  
  address: {  
    city: "New York",  
    zip: "10001"  
  }  
}
```

```
let { name, address: { city, zip } } = user
```

```
console.log(name)  
console.log(city)
```

```
console.log(zip)
```

Output:

Alice

New York

10001

Note: address itself is not a variable here. Only city and zip are extracted.

To also get address:

```
let { name, address, address: { city } } = user
```

REST PATTERN

Collect remaining properties:

```
let person = {  
  name: "Alice",  
  age: 25,  
  city: "New York",  
  country: "USA"  
}
```

```
let { name, ...rest } = person
```

```
console.log(name)  
console.log(rest)
```

Output:

Alice

```
{ age: 25, city: "New York", country: "USA" }
```

Useful for separating certain properties from the rest.

FUNCTION PARAMETERS

Destructure in function parameters:

```
function greet({ name, age }) {  
  console.log(`Hello, ${name}! You are ${age}.`)  
}
```

```
let person = { name: "Alice", age: 25, city: "New York" }
```

```
greet(person)
```

Output: Hello, Alice! You are 25.

Only extract what you need.

With defaults:

```
function greet({ name = "Guest", age = 0 } = {}) {  
  console.log(`Hello, ${name}! You are ${age}.`)  
}
```

```
greet({ name: "Alice" })  
greet({})  
greet()
```

Output:

Hello, Alice! You are 0.

Hello, Guest! You are 0.

Hello, Guest! You are 0.

The `= {}` provides a default empty object if nothing is passed.

PRACTICAL EXAMPLES

API response handling:

```
let response = {  
  status: 200,  
  data: {  
    user: {  
      id: 1,  
      name: "Alice",  
      email: "alice@example.com"  
    }  
  },  
  headers: {  
    contentType: "application/json"  
  }  
}
```

```
let { status, data: { user: { name, email } } } = response
```

```
console.log(`Status: ${status}`)  
console.log(`User: ${name} (${email})`)
```

Configuration options:

```
function createButton({ text = "Click", color = "blue", size =  
"medium", disabled = false } = {}) {  
  console.log(` Button: ${text}`)  
  console.log(` Style: ${color}, ${size}`)  
  console.log(` Disabled: ${disabled}`)  
}
```

```
createButton({ text: "Submit", color: "green" })
```

Output:

Button: Submit

Style: green, medium

Disabled: false

Event handling:

```
function handleEvent({ type, target, timestamp }) {
```

```
console.log(`Event: ${type}`)  
console.log(`Target: ${target}`)  
console.log(`Time: ${timestamp}`)  
}
```

```
let event = {  
  type: "click",  
  target: "button",  
  timestamp: Date.now(),  
  bubbles: true  
}
```

handleEvent(event)

Swap properties:

```
let point = { x: 10, y: 20 }
```

```
let { x: y, y: x } = point
```

```
let swapped = { x, y }
```

```
console.log(swapped)
```

Output: { x: 20, y: 10 }

Remove properties:

```
let user = {  
  name: "Alice",  
  password: "secret",  
  email: "alice@example.com"  
}
```

```
let { password, ...safeUser } = user
```

```
console.log(safeUser)
```

Output: { name: "Alice", email: "alice@example.com" }

DESTRUCTURING IN LOOPS

```
let users = [  
  { name: "Alice", age: 25 },  
  { name: "Bob", age: 30 },  
  { name: "Charlie", age: 35 }  
]  
  
for (let { name, age } of users) {  
  console.log(`${name} is ${age}`)  
}
```

Output:

Alice is 25

Bob is 30

Charlie is 35

With array methods:

```
users.forEach(({ name }) => {  
  console.log(name)  
})  
  
let names = users.map(({ name }) => name)
```

COMPLEX DESTRUCTURING

```
let data = {  
  users: [  
    { id: 1, name: "Alice", roles: ["admin", "user"] },  
    { id: 2, name: "Bob", roles: ["user"] }  
  ],  
  meta: {  
    total: 2,  
  },  
}
```

```
page: 1
}  
}
```

```
let {  
  users: [firstUser, secondUser],  
  meta: { total }  
} = data
```

```
console.log(firstUser.name)  
console.log(total)
```

Output:
Alice
2

PRACTICE

Destructure name and age from a person object.

Rename a property while destructuring.

Provide a default value for a missing property.

Destructure a nested object.

Use rest to collect remaining properties.

WHAT YOU LEARNED

You learned to extract properties with `{ name } = object`.

You learned to rename with `{ name: newName }`.

You learned to provide defaults with `{ name = "default" }`.

You learned nested destructuring.

You learned the rest pattern to collect remaining properties.
You learned to destructure in function parameters.

The next chapter teaches you about constructor functions and classes.

BOOK 6: OBJECTS

Chapter 6: Constructors and Classes

CREATING MULTIPLE SIMILAR OBJECTS

So far you created objects one at a time:

```
let user1 = { name: "Alice", age: 25 }  
let user2 = { name: "Bob", age: 30 }  
let user3 = { name: "Charlie", age: 35 }
```

This is repetitive. What if you need 100 users?

CONSTRUCTOR FUNCTIONS

A constructor function creates objects:

```
function User(name, age) {  
  this.name = name  
  this.age = age  
  this.greet = function() {  
    console.log(`Hello, I am ${this.name}`)  
  }  
}
```

```
let user1 = new User("Alice", 25)  
let user2 = new User("Bob", 30)
```

```
console.log(user1.name)
console.log(user2.name)
user1.greet()
```

Output:

Alice

Bob

Hello, I am Alice

The new keyword:

1. Creates a new empty object
2. Sets this to that object
3. Runs the constructor function
4. Returns the object

Constructor function names start with a capital letter by convention.

CLASSES (MODERN SYNTAX)

ES6 introduced class syntax:

```
class User {
  constructor(name, age) {
    this.name = name
    this.age = age
  }

  greet() {
    console.log(`Hello, I am ${this.name}`)
  }
}
```

```
let user1 = new User("Alice", 25)
let user2 = new User("Bob", 30)
```

```
user1.greet()
```

```
user2.greet()
```

Output:

Hello, I am Alice

Hello, I am Bob

Classes are cleaner. The constructor method runs when you use new. Other methods are added to the prototype.

CLASS PROPERTIES

Properties defined in constructor:

```
class Product {  
  constructor(name, price) {  
    this.name = name  
    this.price = price  
    this.inStock = true  
  }  
}
```

Class fields (modern):

```
class Product {  
  inStock = true  
  discount = 0  
  
  constructor(name, price) {  
    this.name = name  
    this.price = price  
  }  
}
```

```
let product = new Product("Laptop", 999)
```

```
console.log(product.inStock)
```

Output: true

CLASS METHODS

```
class Calculator {  
  constructor(value = 0) {  
    this.value = value  
  }
```

```
  add(num) {  
    this.value += num  
    return this  
  }
```

```
  subtract(num) {  
    this.value -= num  
    return this  
  }
```

```
  multiply(num) {  
    this.value *= num  
    return this  
  }
```

```
  getResult() {  
    return this.value  
  }  
}
```

```
let calc = new Calculator(10)
```

```
let result = calc.add(5).multiply(2).subtract(3).getResult()
```

```
console.log(result)
```

Output: 27

Returning this allows method chaining.

GETTERS AND SETTERS

```
class Circle {  
  constructor(radius) {  
    this._radius = radius  
  }  
  
  get radius() {  
    return this._radius  
  }  
  
  set radius(value) {  
    if (value < 0) {  
      throw new Error("Radius cannot be negative")  
    }  
    this._radius = value  
  }  
  
  get diameter() {  
    return this._radius * 2  
  }  
  
  get area() {  
    return Math.PI * this._radius * this._radius  
  }  
}  
  
let circle = new Circle(5)  
  
console.log(circle.radius)  
console.log(circle.diameter)  
console.log(circle.area)  
  
circle.radius = 10
```

```
console.log(circle.diameter)
```

Output:

5

10

78.54...

20

Getters and setters look like properties but run methods.

STATIC METHODS

Methods on the class itself, not instances:

```
class MathHelper {  
  static add(a, b) {  
    return a + b  
  }
```

```
  static multiply(a, b) {  
    return a * b  
  }
```

```
  static random(min, max) {  
    return Math.floor(Math.random() * (max - min + 1)) + min  
  }  
}
```

```
console.log(MathHelper.add(5, 3))  
console.log(MathHelper.random(1, 10))
```

Output:

8

(random number 1-10)

Static methods are called on the class, not on instances.

```
let helper = new MathHelper()
helper.add(5, 3)
```

Error. Static methods are not on instances.

STATIC PROPERTIES

```
class Config {
  static version = "1.0.0"
  static appName = "MyApp"
  static maxUsers = 100
}

console.log(Config.version)
console.log(Config.appName)
```

Output:

1.0.0

MyApp

PRIVATE FIELDS

Use # prefix for private:

```
class BankAccount {
  #balance = 0

  constructor(initialBalance) {
    this.#balance = initialBalance
  }

  deposit(amount) {
    if (amount > 0) {
      this.#balance += amount
    }
  }
}
```

```
}  
}
```

```
withdraw(amount) {  
  if (amount > 0 && amount <= this.#balance) {  
    this.#balance -= amount  
    return true  
  }  
  return false  
}
```

```
getBalance() {  
  return this.#balance  
}  
}
```

```
let account = new BankAccount(1000)
```

```
account.deposit(500)
```

```
console.log(account.getBalance())
```

```
console.log(account.#balance)
```

Output:

1500

Error: Private field

Private fields cannot be accessed outside the class.

PRACTICAL EXAMPLES

Todo Item:

```
class TodoItem {  
  #completed = false
```

```
constructor(text, priority = 1) {  
  this.text = text  
  this.priority = priority  
  this.createdAt = new Date()  
}
```

```
complete() {  
  this.#completed = true  
  this.completedAt = new Date()  
}
```

```
isCompleted() {  
  return this.#completed  
}
```

```
toString() {  
  let status = this.#completed ? "DONE" : "TODO"  
  return `[${status}] ${this.text}`  
}  
}
```

```
let task = new TodoItem("Learn JavaScript", 1)
```

```
console.log(task.toString())  
task.complete()  
console.log(task.toString())
```

Player:

```
class Player {  
  static maxLevel = 100
```

```
  constructor(name) {  
    this.name = name  
    this.level = 1  
    this.experience = 0  
    this.health = 100  
  }
```

```
  gainExperience(amount) {
```

```

this.experience += amount
while (this.experience >= this.experienceNeeded() && this.level
< Player.maxLevel) {
  this.levelUp()
}
}

```

```

experienceNeeded() {
  return this.level * 100
}

```

```

levelUp() {
  this.level ++
  this.experience -= this.experienceNeeded()
  this.health += 10
  console.log(`${this.name} leveled up to ${this.level}!`)
}
}

```

```

let player = new Player("Hero")
player.gainExperience(250)

```

Shopping Cart:

```

class ShoppingCart {
  #items = []

  addItem(product, quantity = 1) {
    let existing = this.#items.find(item => item.product.id ===
product.id)
    if (existing) {
      existing.quantity += quantity
    } else {
      this.#items.push({ product, quantity })
    }
  }
}

```

```

getTotal() {
  return this.#items.reduce((total, item) => {
    return total + (item.product.price * item.quantity)
  })
}

```

```
}, 0)  
}
```

```
getItemCount() {  
  return this.#items.reduce((count, item) => count +  
    item.quantity, 0)  
}
```

```
getItems() {  
  return [...this.#items]  
}  
}
```

PRACTICE

Create a class representing a book with title and author.

Add a method to the class.

Create a getter for a calculated property.

Add a static method to the class.

Create a class with a private field.

WHAT YOU LEARNED

You learned constructor functions create objects.

You learned class syntax is cleaner than constructor functions.

You learned about constructor, methods, and properties.

You learned getters and setters for computed properties.

You learned static methods and properties belong to the class.

You learned private fields with # prefix.

The next chapter teaches you about inheritance.

BOOK 6: OBJECTS

Chapter 7: Inheritance

WHAT IS INHERITANCE

Inheritance lets one class use properties and methods from another:

```
class Animal {  
  constructor(name) {  
    this.name = name  
  }  
  
  speak() {  
    console.log(`${this.name} makes a sound`)  
  }  
}  
  
class Dog extends Animal {  
  bark() {  
    console.log(`${this.name} barks`)  
  }  
}  
  
let dog = new Dog("Rex")  
dog.speak()  
dog.bark()
```

Output:

Rex makes a sound

Rex barks

Dog extends Animal. Dog inherits name and speak from Animal, and adds bark.

THE EXTENDS KEYWORD

```
class Parent {  
  parentMethod() {  
    console.log("Parent method")  
  }  
}
```

```
class Child extends Parent {  
  childMethod() {  
    console.log("Child method")  
  }  
}
```

```
let child = new Child()  
child.parentMethod()  
child.childMethod()
```

Output:

Parent method

Child method

Child has both its own methods and Parent's methods.

THE SUPER KEYWORD

super calls the parent's constructor:

```
class Animal {  
  constructor(name) {  
    this.name = name  
  }  
}
```

```
class Dog extends Animal {
```

```
constructor(name, breed) {  
  super(name)  
  this.breed = breed  
}  
}
```

```
let dog = new Dog("Rex", "German Shepherd")
```

```
console.log(dog.name)  
console.log(dog.breed)
```

Output:

Rex

German Shepherd

`super(name)` calls `Animal`'s constructor with the name.

If a child class has a constructor, it must call `super()` before using `this`.

OVERRIDING METHODS

Child classes can override parent methods:

```
class Animal {  
  speak() {  
    console.log("Animal speaks")  
  }  
}
```

```
class Dog extends Animal {  
  speak() {  
    console.log("Dog barks")  
  }  
}
```

```
class Cat extends Animal {
```

```
    speak() {  
      console.log("Cat meows")  
    }  
  }  
  
let animal = new Animal()  
let dog = new Dog()  
let cat = new Cat()
```

```
animal.speak()  
dog.speak()  
cat.speak()
```

Output:
Animal speaks
Dog barks
Cat meows

Each class has its own implementation of speak.

CALLING PARENT METHODS

Use `super.method()` to call the parent's version:

```
class Animal {  
  speak() {  
    console.log(`${this.name} makes a sound`)  
  }  
}
```

```
class Dog extends Animal {  
  constructor(name) {  
    super(name)  
    this.name = name  
  }  
  
  speak() {
```

```
super.speak()  
console.log(`${this.name} also barks!`)  
}  
}
```

```
let dog = new Dog("Rex")  
dog.speak()
```

Output:

Rex makes a sound
Rex also barks!

The child calls the parent method and adds more behavior.

INSTANCEOF

Check if an object is an instance of a class:

```
class Animal {}  
class Dog extends Animal {}  
class Cat extends Animal {}
```

```
let dog = new Dog()
```

```
console.log(dog instanceof Dog)  
console.log(dog instanceof Animal)  
console.log(dog instanceof Cat)
```

Output:

true
true
false

dog is a Dog. Dog extends Animal, so dog is also an Animal.

PRACTICAL EXAMPLES

Shape hierarchy:

```
class Shape {  
  constructor(color) {  
    this.color = color  
  }  

```

```
  describe() {  
    return `A ${this.color} shape`  
  }  

```

```
  getArea() {  
    return 0  
  }  
}
```

```
class Rectangle extends Shape {  
  constructor(color, width, height) {  
    super(color)  
    this.width = width  
    this.height = height  
  }  

```

```
  describe() {  
    return `A ${this.color} rectangle (${this.width}x${this.height})`  
  }  

```

```
  getArea() {  
    return this.width * this.height  
  }  
}
```

```
class Circle extends Shape {  
  constructor(color, radius) {  
    super(color)  
    this.radius = radius  
  }  

```

```
describe() {  
  return `A ${this.color} circle (radius ${this.radius})`  
}
```

```
getArea() {  
  return Math.PI * this.radius * this.radius  
}  
}
```

```
let shapes = [  
  new Rectangle("red", 10, 5),  
  new Circle("blue", 7),  
  new Rectangle("green", 4, 4)  
]
```

```
for (let shape of shapes) {  
  console.log(shape.describe())  
  console.log(` Area: ${shape.getArea().toFixed(2)}`)  
  console.log("")  
}
```

Vehicle hierarchy:

```
class Vehicle {  
  constructor(brand, model) {  
    this.brand = brand  
    this.model = model  
    this.speed = 0  
  }
```

```
  accelerate(amount) {  
    this.speed += amount  
    console.log(`${this.brand} ${this.model} accelerated to  
    ${this.speed} mph`)  
  }
```

```
  brake(amount) {  
    this.speed = Math.max(0, this.speed - amount)  
    console.log(`${this.brand} ${this.model} slowed to ${this.speed}`)
```

```
mph`)  
}  
}
```

```
class Car extends Vehicle {  
  constructor(brand, model, doors) {  
    super(brand, model)  
    this.doors = doors  
  }  
}
```

```
  honk() {  
    console.log("Beep beep!")  
  }  
}
```

```
class Motorcycle extends Vehicle {  
  constructor(brand, model, type) {  
    super(brand, model)  
    this.type = type  
  }  
}
```

```
  wheelie() {  
    if (this.speed > 20) {  
      console.log("Doing a wheelie!")  
    } else {  
      console.log("Need more speed for a wheelie")  
    }  
  }  
}
```

```
let car = new Car("Toyota", "Camry", 4)  
let bike = new Motorcycle("Honda", "CBR", "Sport")
```

```
car.accelerate(50)  
car.honk()
```

```
bike.accelerate(30)  
bike.wheelie()
```

User roles:

```
class User {
  constructor(name, email) {
    this.name = name
    this.email = email
  }

  getPermissions() {
    return ["read"]
  }

  describe() {
    return `${this.name} (${this.email})`
  }
}

class Admin extends User {
  constructor(name, email, department) {
    super(name, email)
    this.department = department
  }

  getPermissions() {
    return [...super.getPermissions(), "write", "delete", "admin"]
  }

  describe() {
    return `Admin: ${super.describe()} - ${this.department}`
  }
}

class Moderator extends User {
  getPermissions() {
    return [...super.getPermissions(), "write", "moderate"]
  }
}

let user = new User("Alice", "alice@example.com")
let admin = new Admin("Bob", "bob@example.com", "IT")
let mod = new Moderator("Charlie", "charlie@example.com")
```

```
console.log(user.describe())
console.log(user.getPermissions())
console.log("")
console.log(admin.describe())
console.log(admin.getPermissions())
```

MULTI-LEVEL INHERITANCE

```
class Animal {
  constructor(name) {
    this.name = name
  }

  eat() {
    console.log(`${this.name} is eating`)
  }
}

class Mammal extends Animal {
  feedMilk() {
    console.log(`${this.name} feeds milk to young`)
  }
}

class Dog extends Mammal {
  bark() {
    console.log(`${this.name} barks`)
  }
}

let dog = new Dog("Rex")
dog.eat()
dog.feedMilk()
dog.bark()
```

Output:

Rex is eating
Rex feeds milk to young
Rex barks

Dog inherits from Mammal, which inherits from Animal.

PRACTICE

Create a base class and a child class that extends it.

Override a method in the child class.

Call the parent method from the child using super.

Create a hierarchy with three levels.

Use isinstance to check object types.

WHAT YOU LEARNED

You learned extends creates a child class.

You learned super() calls the parent constructor.

You learned super.method() calls parent methods.

You learned child classes can override methods.

You learned isinstance checks class membership.

You learned to build class hierarchies.

The next chapter teaches you about working with JSON data.

BOOK 6: OBJECTS

Chapter 8: Working with JSON

WHAT IS JSON

JSON (JavaScript Object Notation) is a text format for data:

```
{  
  "name": "Alice",  
  "age": 25,  
  "city": "New York"  
}
```

JSON looks like JavaScript objects but:

- Property names must be in double quotes
- Strings must use double quotes
- No trailing commas
- No functions or undefined

JSON is used to:

- Send data between server and browser
- Store configuration files
- Save application state

JSON VS JAVASCRIPT OBJECTS

JavaScript object:

```
let person = {  
  name: "Alice",  
  age: 25  
}
```

JSON string:

```
let json = '{"name": "Alice", "age": 25}'
```

JSON is text. JavaScript objects are in-memory data structures.

JSON.STRINGIFY

Converts an object to a JSON string:

```
let person = {  
  name: "Alice",  
  age: 25,  
  city: "New York"  
}
```

```
let json = JSON.stringify(person)
```

```
console.log(json)  
console.log(typeof json)
```

Output:

```
{"name":"Alice","age":25,"city":"New York"}  
string
```

Pretty printing with indentation:

```
let json = JSON.stringify(person, null, 2)
```

```
console.log(json)
```

Output:

```
{  
  "name": "Alice",  
  "age": 25,  
  "city": "New York"  
}
```

The third argument is the number of spaces for indentation.

JSON.PARSE

Converts a JSON string to an object:

```
let json = '{"name": "Alice", "age": 25}'
```

```
let person = JSON.parse(json)
```

```
console.log(person.name)
```

```
console.log(person.age)
```

```
console.log(typeof person)
```

Output:

Alice

25

object

Parse errors:

```
let badJson = '{name: "Alice"}'
```

```
try {
```

```
  JSON.parse(badJson)
```

```
} catch (error) {
```

```
  console.log("Invalid JSON:", error.message)
```

```
}
```

Property names must be quoted in JSON.

WHAT JSON SUPPORTS

Supported types:

- Strings: "hello"

- Numbers: 42, 3.14

- Booleans: true, false

- null

- Arrays: [1, 2, 3]

- Objects: {"key": "value"}

Not supported:

- undefined
- Functions
- Dates (become strings)
- Maps and Sets
- Symbols

HANDLING DATES

Dates become strings:

```
let event = {  
  title: "Meeting",  
  date: new Date("2024-03-15")  
}
```

```
let json = JSON.stringify(event)
```

```
console.log(json)
```

Output: {"title":"Meeting","date":"2024-03-15T00:00:00.000Z"}

Parse back to Date:

```
let parsed = JSON.parse(json)
```

```
console.log(typeof parsed.date)
```

```
parsed.date = new Date(parsed.date)
```

```
console.log(typeof parsed.date)
```

Output:

string

object

You must manually convert date strings back to Date objects.

REPLACER FUNCTION

Customize what gets stringified:

```
let user = {  
  name: "Alice",  
  password: "secret",  
  email: "alice@example.com"  
}
```

```
let json = JSON.stringify(user, (key, value) => {  
  if (key === "password") {  
    return undefined  
  }  
  return value  
})
```

```
console.log(json)
```

Output: {"name":"Alice","email":"alice@example.com"}

The password is excluded.

Select specific properties:

```
let json = JSON.stringify(user, ["name", "email"])
```

```
console.log(json)
```

Output: {"name":"Alice","email":"alice@example.com"}

REVIVER FUNCTION

Customize how values are parsed:

```
let json = '{"name": "Alice", "date": "2024-03-15"}'
```

```
let data = JSON.parse(json, (key, value) => {  
  if (key === "date") {  
    return new Date(value)  
  }  
  return value  
})
```

```
console.log(data.date instanceof Date)
```

Output: true

The reviver transforms the date string to a Date object.

PRACTICAL EXAMPLES

Save to localStorage:

```
let settings = {  
  theme: "dark",  
  language: "en",  
  notifications: true  
}
```

```
localStorage.setItem("settings", JSON.stringify(settings))
```

```
let loaded = JSON.parse(localStorage.getItem("settings"))
```

```
console.log(loaded.theme)
```

API response:

```
let response = `{
  "status": "success",
  "data": {
    "users": [
      {"id": 1, "name": "Alice"},
      {"id": 2, "name": "Bob"}
    ]
  }
}`
```

```
let result = JSON.parse(response)
```

```
result.data.users.forEach(user => {
  console.log(`${user.id}: ${user.name}`)
})
```

Deep copy:

```
let original = {
  name: "Alice",
  address: {
    city: "New York"
  }
}
```

```
let copy = JSON.parse(JSON.stringify(original))
```

```
copy.address.city = "Boston"
```

```
console.log(original.address.city)
console.log(copy.address.city)
```

Output:

```
New York
Boston
```

Configuration file:

```
let config = `{
  "apiUrl": "https://api.example.com",
```

```
"timeout": 5000,  
"retries": 3,  
"features": {  
  "darkMode": true,  
  "analytics": false  
}  
}`
```

```
let settings = JSON.parse(config)
```

```
console.log(settings.apiUrl)  
console.log(settings.features.darkMode)
```

HANDLING ERRORS

Always wrap parse in try-catch:

```
function safeParseJSON(text, fallback = null) {  
  try {  
    return JSON.parse(text)  
  } catch (error) {  
    console.error("JSON parse error:", error.message)  
    return fallback  
  }  
}
```

```
let data = safeParseJSON('{ "valid": true }')  
console.log(data)
```

```
let bad = safeParseJSON('not json', {})  
console.log(bad)
```

Output:

```
{ valid: true }  
{ }
```

TOSTRING METHOD

Custom serialization with toJSON:

```
class User {  
  constructor(name, password) {  
    this.name = name  
    this.password = password  
  }  
  
  toJSON() {  
    return {  
      name: this.name  
    }  
  }  
}  
  
let user = new User("Alice", "secret")  
let json = JSON.stringify(user)  
  
console.log(json)
```

Output: {"name":"Alice"}

toJSON controls what gets serialized.

PRACTICE

Convert an object to a JSON string.

Parse a JSON string into an object.

Pretty-print JSON with indentation.

Exclude a property when stringifying.

Handle a parse error with try-catch.

WHAT YOU LEARNED

You learned JSON is a text format for data exchange.

You learned `JSON.stringify` converts objects to strings.

You learned `JSON.parse` converts strings to objects.

You learned JSON limitations (no functions, dates become strings).

You learned to use `replacer` and `reviver` functions.

You learned to handle JSON errors.

The next chapter teaches you about object patterns.

BOOK 6: OBJECTS

Chapter 9: Object Patterns

FACTORY PATTERN

A function that creates objects:

```
function createUser(name, email) {  
  return {  
    name,  
    email,  
    isActive: true,  
    greet() {  
      console.log(`Hello, I am ${this.name}`)  
    }  
  }  
}
```

```
let user1 = createUser("Alice", "alice@example.com")  
let user2 = createUser("Bob", "bob@example.com")
```

```
user1.greet()
```

Output: Hello, I am Alice

Factory functions are simpler than classes for creating objects.

FACTORY WITH OPTIONS

```
function createButton(options = {}) {  
  let defaults = {  
    text: "Click me",  
    color: "blue",  
    size: "medium",  
    disabled: false  
  }  
  
  return {  
    ...defaults,  
    ...options,  
    click() {  
      if (!this.disabled) {  
        console.log(` Button "${this.text}" clicked`)  
      }  
    }  
  }  
}
```

```
let btn1 = createButton()  
let btn2 = createButton({ text: "Submit", color: "green" })  
  
btn1.click()  
btn2.click()
```

SINGLETON PATTERN

Only one instance exists:

```
const Config = {  
  apiUrl: "https://api.example.com",  
  timeout: 5000,  
  retries: 3,
```

```
  get(key) {  
    return this[key]  
  },
```

```
  set(key, value) {  
    this[key] = value  
  }  
}
```

```
console.log(Config.get("apiUrl"))  
Config.set("timeout", 10000)  
console.log(Config.get("timeout"))
```

```
Object.freeze(Config)
```

Config is a single shared instance.

Using closure for private data:

```
const Logger = (function() {  
  let logs = []  
  
  return {  
    log(message) {  
      let entry = { time: new Date(), message }  
      logs.push(entry)  
      console.log(` [LOG] ${message}`)  
    },
```

```
    getLogs() {  
      return [...logs]  
    }  
  }  
})()
```

```
},
```

```
clear() {  
  logs = []  
}  
}  
})()
```

```
Logger.log("Application started")  
Logger.log("User logged in")
```

```
console.log(Logger.getLogs().length)
```

MODULE PATTERN

Organize related code together:

```
const Calculator = (function() {  
  let history = []  
  
  function record(operation, result) {  
    history.push({ operation, result, time: new Date() })  
  }  
  
  return {  
    add(a, b) {  
      let result = a + b  
      record(`${a} + ${b}`, result)  
      return result  
    },  
  
    subtract(a, b) {  
      let result = a - b  
      record(`${a} - ${b}`, result)  
      return result  
    },  
  }  
})()
```

```
getHistory() {  
  return [...history]  
}  
}  
})()
```

```
console.log(Calculator.add(5, 3))  
console.log(Calculator.subtract(10, 4))  
console.log(Calculator.getHistory())
```

Private variables (history) are hidden inside the closure.

BUILDER PATTERN

Construct objects step by step:

```
class QueryBuilder {  
  constructor(table) {  
    this.table = table  
    this.conditions = []  
    this.orderBy = null  
    this.limitCount = null  
  }  
  
  where(condition) {  
    this.conditions.push(condition)  
    return this  
  }  
  
  order(field, direction = "ASC") {  
    this.orderBy = { field, direction }  
    return this  
  }  
  
  limit(count) {  
    this.limitCount = count  
    return this  
  }  
}
```

```

}

build() {
let query = `SELECT * FROM ${this.table}`

if (this.conditions.length > 0) {
query += ` WHERE ${this.conditions.join(" AND ")} `
}

if (this.orderBy) {
query += ` ORDER BY ${this.orderBy.field}
${this.orderBy.direction}`
}

if (this.limitCount) {
query += ` LIMIT ${this.limitCount}`
}

return query
}
}

let query = new QueryBuilder("users")
  .where("age > 18")
  .where("active = true")
  .order("name")
  .limit(10)
  .build()

console.log(query)

```

Output: SELECT * FROM users WHERE age > 18 AND active = true ORDER BY name ASC LIMIT 10

OBSERVER PATTERN

Objects subscribe to events:

```

class EventEmitter {
  constructor() {
    this.listeners = {}
  }

  on(event, callback) {
    if (!this.listeners[event]) {
      this.listeners[event] = []
    }
    this.listeners[event].push(callback)
  }

  off(event, callback) {
    if (this.listeners[event]) {
      this.listeners[event] = this.listeners[event]
        .filter(cb => cb !== callback)
    }
  }

  emit(event, data) {
    if (this.listeners[event]) {
      this.listeners[event].forEach(callback => callback(data))
    }
  }
}

let events = new EventEmitter()

events.on("userLogin", (user) => {
  console.log(`Welcome, ${user.name}!`)
})

events.on("userLogin", (user) => {
  console.log(`Logging login for ${user.name}`)
})

events.emit("userLogin", { name: "Alice" })

```

Output:

Welcome, Alice!
Logging login for Alice

PROXY PATTERN

Wrap an object to control access:

```
function createProxy(target) {  
  return {  
    get(prop) {  
      console.log(`Getting ${prop}`)  
      return target[prop]  
    },  
  
    set(prop, value) {  
      console.log(`Setting ${prop} to ${value}`)  
      target[prop] = value  
    }  
  }  
}  
  
let user = { name: "Alice", age: 25 }  
let proxy = createProxy(user)  
  
proxy.get("name")  
proxy.set("age", 26)
```

Using JavaScript Proxy:

```
let user = { name: "Alice", age: 25 }  
  
let handler = {  
  get(target, prop) {  
    console.log(`Accessing ${prop}`)  
    return target[prop]  
  },  
}
```

```
set(target, prop, value) {  
  console.log(`Setting ${prop} = ${value}`)  
  target[prop] = value  
  return true  
}  
}
```

```
let userProxy = new Proxy(user, handler)
```

```
console.log(userProxy.name)  
userProxy.age = 26
```

MIXIN PATTERN

Add functionality to objects:

```
let canFly = {  
  fly() {  
    console.log(`${this.name} is flying`)  
  }  
}
```

```
let canSwim = {  
  swim() {  
    console.log(`${this.name} is swimming`)  
  }  
}
```

```
class Animal {  
  constructor(name) {  
    this.name = name  
  }  
}
```

```
class Duck extends Animal {}
```

```
Object.assign(Duck.prototype, canFly, canSwim)
```

```
let duck = new Duck("Donald")
duck.fly()
duck.swim()
```

Output:

Donald is flying

Donald is swimming

STATE PATTERN

Object behavior changes based on state:

```
class TrafficLight {
  constructor() {
    this.states = {
      red: {
        color: "red",
        next: "green",
        duration: 5000
      },
      green: {
        color: "green",
        next: "yellow",
        duration: 4000
      },
      yellow: {
        color: "yellow",
        next: "red",
        duration: 1000
      }
    }
    this.currentState = "red"
  }
}
```

```
getColor() {
  return this.states[this.currentState].color
}
```

```
}

change() {
  let current = this.states[this.currentState]
  this.currentState = current.next
  console.log(`Light changed to ${this.getColor()}`)
}
}
```

```
let light = new TrafficLight()
console.log(light.getColor())
light.change()
light.change()
light.change()
```

Output:

red

Light changed to green

Light changed to yellow

Light changed to red

PRACTICE

Create a factory function for products.

Implement a singleton for application settings.

Create a builder for constructing complex objects.

Implement a simple event emitter.

Create a mixin to add functionality to objects.

WHAT YOU LEARNED

You learned factory functions create objects.
You learned singletons have one shared instance.
You learned the module pattern for organization and privacy.
You learned the builder pattern for step-by-step construction.
You learned the observer pattern for event handling.
You learned mixins add functionality to objects.

The next chapter brings everything together with practice programs.

BOOK 6: OBJECTS

Chapter 10: Practice Object Programs

GAME CHARACTER SYSTEM

Complete RPG character with items and abilities:

```
class Character {
  constructor(name, characterClass) {
    this.name = name
    this.characterClass = characterClass
    this.level = 1
    this.experience = 0
    this.stats = this.getBaseStats()
    this.health = this.stats.maxHealth
    this.mana = this.stats.maxMana
    this.inventory = []
    this.equipment = {}
    this.abilities = []
  }

  getBaseStats() {
    let baseStats = {
      warrior: { maxHealth: 120, maxMana: 30, attack: 15, defense: 10 },
      mage: { maxHealth: 70, maxMana: 100, attack: 8, defense: 5 },
      rogue: { maxHealth: 90, maxMana: 50, attack: 12, defense: 7 }
    }
```

```

}
return { ...baseStats[this.characterClass] }
}

attack(target) {
let damage = this.stats.attack + (this.equipment.weapon?.attack
|| 0)
let actualDamage = Math.max(1, damage - target.stats.defense)
target.health -= actualDamage
console.log(` ${this.name} attacks ${target.name} for
${actualDamage} damage`)

if (target.health <= 0) {
this.gainExperience(target.level * 50)
}
}

gainExperience(amount) {
this.experience += amount
console.log(` ${this.name} gained ${amount} XP`)

while (this.experience >= this.experienceNeeded()) {
this.levelUp()
}
}

experienceNeeded() {
return this.level * 100
}

levelUp() {
this.level++
this.experience -= this.experienceNeeded()
this.stats.maxHealth += 10
this.stats.maxMana += 5
this.stats.attack += 2
this.stats.defense += 1
this.health = this.stats.maxHealth
this.mana = this.stats.maxMana
console.log(` ${this.name} reached level ${this.level}!`)
}

```

```
}
```

```
equip(item) {  
  this.equipment[item.slot] = item  
  console.log(`${this.name} equipped ${item.name}`)  
}
```

```
addToInventory(item) {  
  this.inventory.push(item)  
  console.log(`${this.name} picked up ${item.name}`)  
}
```

```
getStatus() {  
  return {  
    name: this.name,  
    class: this.characterClass,  
    level: this.level,  
    health: `${this.health}/${this.stats.maxHealth}`,  
    mana: `${this.mana}/${this.stats.maxMana}`,  
    attack: this.stats.attack,  
    defense: this.stats.defense  
  }  
}  
}
```

```
let hero = new Character("Arthas", "warrior")  
let enemy = new Character("Goblin", "rogue")
```

```
console.log("== CHARACTER SYSTEM ==")  
console.log(hero.getStatus())
```

```
hero.addToInventory({ name: "Health Potion", type: "consumable"  
})
```

```
hero.equip({ name: "Iron Sword", slot: "weapon", attack: 5 })
```

```
hero.attack(enemy)  
hero.attack(enemy)
```

BANK ACCOUNT SYSTEM

Complete banking with transactions:

```
class BankAccount {
  #balance
  #transactions
  #accountNumber

  static #nextAccountNumber = 1000

  constructor(owner, initialDeposit = 0) {
    this.#accountNumber = BankAccount.#nextAccountNumber + +
    this.owner = owner
    this.#balance = 0
    this.#transactions = []
    this.createdAt = new Date()

    if (initialDeposit > 0) {
      this.deposit(initialDeposit, "Initial deposit")
    }
  }

  get accountNumber() {
    return this.#accountNumber
  }

  get balance() {
    return this.#balance
  }

  #recordTransaction(type, amount, description, balanceAfter) {
    this.#transactions.push({
      type,
      amount,
      description,
      balanceAfter,
      timestamp: new Date()
    })
  }
}
```

```
}
```

```
deposit(amount, description = "Deposit") {  
  if (amount <= 0) {  
    throw new Error("Deposit amount must be positive")  
  }  
}
```

```
  this.#balance += amount  
  this.#recordTransaction("deposit", amount, description,  
this.#balance)  
  console.log(`Deposited ${amount}. New balance: $  
${this.#balance}`)  
  return true  
}
```

```
withdraw(amount, description = "Withdrawal") {  
  if (amount <= 0) {  
    throw new Error("Withdrawal amount must be positive")  
  }  
}
```

```
  if (amount > this.#balance) {  
    console.log("Insufficient funds")  
    return false  
  }
```

```
  this.#balance -= amount  
  this.#recordTransaction("withdrawal", amount, description,  
this.#balance)  
  console.log(`Withdrew ${amount}. New balance: $  
${this.#balance}`)  
  return true  
}
```

```
transfer(toAccount, amount) {  
  if (this.withdraw(amount, `Transfer to account  
${toAccount.accountNumber}`)) {  
    toAccount.deposit(amount, `Transfer from account  
${this.#accountNumber}`)  
    return true  
  }  
}
```

```
return false
}
```

```
getStatement() {
  console.log(`\n=== Account Statement ===`)
  console.log(`Account: ${this.#accountNumber}`)
  console.log(`Owner: ${this.owner}`)
  console.log(`\nTransactions:`)

  this.#transactions.forEach(t => {
    let sign = t.type === "deposit" ? "+" : "-"
    console.log(`  ${sign} ${t.amount} - ${t.description}`)
  })

  console.log(`\nCurrent Balance: $$${this.#balance}`)
}
}
```

```
console.log("=== BANK SYSTEM ===")
```

```
let account1 = new BankAccount("Alice", 1000)
let account2 = new BankAccount("Bob", 500)
```

```
account1.deposit(250)
account1.withdraw(100)
account1.transfer(account2, 200)
```

```
account1.getStatement()
account2.getStatement()
```

TODO APPLICATION

Complete task management:

```
class TodoApp {
  constructor() {
    this.todos = []
  }
}
```

```
this.nextId = 1
this.filters = {
  all: () => true,
  active: todo => !todo.completed,
  completed: todo => todo.completed
}
}
```

```
addTodo(text, priority = "normal") {
  let todo = {
    id: this.nextId++,
    text,
    priority,
    completed: false,
    createdAt: new Date(),
    completedAt: null
  }
  this.todos.push(todo)
  console.log(`Added: "${text}"`)
  return todo
}
```

```
completeTodo(id) {
  let todo = this.todos.find(t => t.id === id)
  if (todo) {
    todo.completed = true
    todo.completedAt = new Date()
    console.log(`Completed: "${todo.text}"`)
    return true
  }
  return false
}
```

```
deleteTodo(id) {
  let index = this.todos.findIndex(t => t.id === id)
  if (index !== -1) {
    let [removed] = this.todos.splice(index, 1)
    console.log(`Deleted: "${removed.text}"`)
    return true
  }
}
```

```
return false
}
```

```
editTodo(id, newText) {
let todo = this.todos.find(t => t.id === id)
if (todo) {
  todo.text = newText
  return true
}
return false
}
```

```
getTodos(filter = "all") {
let filterFn = this.filters[filter] || this.filters.all
return this.todos.filter(filterFn)
}
```

```
getStats() {
let total = this.todos.length
let completed = this.todos.filter(t => t.completed).length
let active = total - completed

return { total, completed, active }
}
```

```
sortByPriority() {
let priorityOrder = { high: 0, normal: 1, low: 2 }
this.todos.sort((a, b) => priorityOrder[a.priority] -
priorityOrder[b.priority])
}
```

```
displayTodos(filter = "all") {
console.log(`\n=== ${filter.toUpperCase()} TODOS ===`)
let todos = this.getTodos(filter)
```

```
if (todos.length === 0) {
  console.log("No todos found")
  return
}
```

```

todos.forEach(todo => {
  let status = todo.completed ? "X" : " "
  let priority = todo.priority !== "normal" ? `[${todo.priority}]` :
  ""

  console.log(`[${status}] ${todo.id}. ${todo.text}${priority}`)
})

let stats = this.getStats()
console.log(`\nTotal: ${stats.total} | Active: ${stats.active} |
Completed: ${stats.completed}`)
}
}

console.log("=== TODO APP ===")

let app = new TodoApp()

app.addToDo("Learn JavaScript", "high")
app.addToDo("Build a project", "normal")
app.addToDo("Write documentation", "low")
app.addToDo("Deploy to production", "high")

app.completeToDo(1)
app.completeToDo(3)

app.displayTodos("all")
app.displayTodos("active")

```

SHOPPING CART WITH DISCOUNTS

E-commerce cart with pricing rules:

```

class Product {
  constructor(id, name, price, category) {
    this.id = id
    this.name = name
    this.price = price
  }
}

```

```
this.category = category
}  
}
```

```
class Discount {  
  constructor(type, value, condition = null) {  
    this.type = type  
    this.value = value  
    this.condition = condition  
  }
```

```
  apply(subtotal, cart) {  
    if (this.condition && !this.condition(subtotal, cart)) {  
      return 0  
    }
```

```
    if (this.type === "percentage") {  
      return subtotal * (this.value / 100)  
    } else if (this.type === "fixed") {  
      return Math.min(this.value, subtotal)  
    }  
    return 0  
  }  
}
```

```
class ShoppingCart {  
  constructor() {  
    this.items = []  
    this.discounts = []  
  }
```

```
  addItem(product, quantity = 1) {  
    let existing = this.items.find(item => item.product.id ===  
    product.id)
```

```
    if (existing) {  
      existing.quantity += quantity  
    } else {  
      this.items.push({ product, quantity })  
    }
```

```
console.log(`Added ${quantity}x ${product.name}`)  
}
```

```
removeItem(productId) {  
  this.items = this.items.filter(item => item.product.id !==  
productId)  
}
```

```
updateQuantity(productId, quantity) {  
  let item = this.items.find(item => item.product.id ===  
productId)  
  if (item) {  
    if (quantity <= 0) {  
      this.removeItem(productId)  
    } else {  
      item.quantity = quantity  
    }  
  }  
}
```

```
addDiscount(discount) {  
  this.discounts.push(discount)  
}
```

```
getSubtotal() {  
  return this.items.reduce((total, item) => {  
    return total + (item.product.price * item.quantity)  
  }, 0)  
}
```

```
getTotalDiscount() {  
  let subtotal = this.getSubtotal()  
  return this.discounts.reduce((total, discount) => {  
    return total + discount.apply(subtotal, this)  
  }, 0)  
}
```

```
getTotal() {  
  return this.getSubtotal() - this.getTotalDiscount()  
}
```

```
}
```

```
checkout() {  
  console.log("\n=== CHECKOUT ===")
```

```
  this.items.forEach(item => {  
    let lineTotal = item.product.price * item.quantity  
    console.log(`${item.product.name} x${item.quantity} @ $  
    ${item.product.price} = $$${lineTotal.toFixed(2)}`)  
  })
```

```
  console.log(`\nSubtotal: $$${this.getSubtotal().toFixed(2)}`)
```

```
  let discount = this.getTotalDiscount()  
  if (discount > 0) {  
    console.log(`Discount: -$$${discount.toFixed(2)}`)  
  }
```

```
  console.log(`Total: $$${this.getTotal().toFixed(2)}`)  
}  
}
```

```
console.log("=== SHOPPING CART ===")
```

```
let laptop = new Product(1, "Laptop", 999.99, "Electronics")  
let mouse = new Product(2, "Mouse", 29.99, "Electronics")  
let keyboard = new Product(3, "Keyboard", 79.99, "Electronics")
```

```
let cart = new ShoppingCart()
```

```
cart.addItem(laptop, 1)  
cart.addItem(mouse, 2)  
cart.addItem(keyboard, 1)
```

```
cart.addDiscount(new Discount("percentage", 10, (subtotal) =>  
  subtotal >= 500))
```

```
cart.checkout()
```

WHAT YOU BUILT

You built four complete programs:

Game character system with stats, equipment, and leveling

Bank account system with transactions and transfers

Todo application with filtering and statistics

Shopping cart with discounts and checkout

Each program demonstrated:

Class design with properties and methods

Private fields for encapsulation

Inheritance and composition

Object patterns from previous chapters

Real-world application logic

WHAT COMES NEXT

You now have mastery of JavaScript fundamentals:

- Variables and data types (Book 1)
- Control flow with decisions (Book 2)
- Loops for repetition (Book 3)
- Functions for organization (Book 4)
- Arrays for collections (Book 5)
- Objects for structured data (Book 6)

These six books cover core JavaScript. Everything else builds on these foundations.

Book 7 teaches you the DOM: how JavaScript interacts with web pages. You will:

- Select and modify HTML elements

- Create and remove elements
- Change styles and attributes
- Build interactive web pages

The DOM is where JavaScript comes alive in the browser.

You finished Book 6. You understand objects and object-oriented programming. Now we make things visual.